ANALYSIS ON AN IMPLEMENTATION OF THE GENS-DOMINGOS SUM-PRODUCT NETWORK STRUCTURAL LEARNING SCHEMA

Renato Lui Geh
Computer Science
Institute of Mathematics and Statistics
University of São Paulo
`renatolg@ime.usp.br`

Abstract. Sum-Product Networks (SPNs) are a class of deep probabilistic graphical models. Inference in them is linear in the number of edges of the graph. Furthermore, exact inference is achieved, in a valid SPN, by running through its edges twice at most, making exact inference linear. The Gens-Domingos SPN Schema is an algorithm for structural learning on such models. In this paper we present an implementation of such schema, analyzing its complexity, discoursing implementational and theoretical details, and finally presenting results and experiments achieved with this implementation.

Keywords `cluster analysis; data mining; probabilistic graphical models; tractable models; machine learning; deep learning`

1. Introduction

A Sum-Product Network (SPN) is a probabilistic graphical model that represents a tractable distribution of probability. If an SPN is valid, then we can perform exact inference in time linear to the graph's edges. Its syntax is different to other conventional models (read bayesian and markov networks) in the sense that its graph does not explicitly model events and (in) dependencies between variables. That is, whilst variables in a bayesian network are represented as nodes in the graph, with each edge connecting two nodes asserting a dependency relationship between the connected variables, a node in an SPN may not necessarily represent a variable or event, neither an edge connecting two nodes represent dependence. In this sense, SPNs can be seen as a type of probabilistic Artificial Neural Network (ANN). However, whilst neural networks represent a function, SPNs model a tractable probability distribution. Furthermore, SPNs are distinct from standard neural networks seeing that, whereas ANNs have only one type of neuron with an activation function mapping to values in $[0,1]$, SPNs have two kinds of neurons, which we will see in the next sections. Still, SPNs retain certain important characteristics from ANNs as we will discuss later, with mainly its deep structure properties [DB11] as the most interesting feature.

The Gens-Domingos Schema [GD13], or `LearnGD` as we will reference it throughout this paper, is an SPN structural learning algorithm proposed by Robert Gens and Pedro Domingos. Gens and Domingos call it a schema because it only provides a template of what the algorithm should be like. We will discuss `LearnGD` in details in the next section. This paper documents a particular implementation of the GD schema. Other implementations may have different results.

In this document, we show how we implemented the `LearnGD` algorithm. We analyze the complexity of each algorithm component in detail, later referring to such analyses when drawing conclusions on the overall complexity of the algorithm. As we have mentioned before, since the `LearnGD` schema heavily depends on implementation, the complexity we achieve in this particular case may differ from other implementations. After each analysis, we then look at the algorithm as a whole, drawing conclusions on time and memory usage, as well as implementation details that could potentially decrease the algorithm runtime. We then show some results on experiments made on image classification and image completion.

2. Sum-Product Networks

In this section we will define SPNs differently from other articles [GD13; PD11; DV12] as the original more convoluted definition is of little use for the `LearnGD` algorithm. Our definition is almost identical to the original `LearnGD` article [GD13], with the exception that we assume that an SPN is already normalized. This fact changes nothing, since Peharz *et al* recently proved that normalized SPNs have as much representability power as unnormalized SPNs [Peh+15]. Before we enunciate the formal definition of an SPN, we will give an informal, vague definition of an SPN in order to explain what completeness, consistency, validity and decomposability — which are an important set of definitions — of an SPN mean.

A sum-product network represents a tractable probability distribution through a DAG. Such digraph must always be weakly connected. A node can either be a leaf, a sum, or a product node. The scope of a node is the set of all variables present in all its descendants. Leaf nodes are tractable probability distributions and their scope is the scope of its distribution, sum nodes represent the summing out of the variables in its scope and product nodes act as feature hierarchy. An edge that has its origin from a sum node has a non-negative weight. We refer to a sub-SPN S rooted at node i as $S(i)$, while the SPN rooted at its root is denoted as $S(\cdot)$ or simply S. The scope of a node will be denoted as $\text{Sc}(i)$, where i is a node. The set of children of a node will be denoted as $\text{Ch}(i)$. Similarly, $\text{Pa}(i)$ is the set of parents of node i.

Definition 2.1 (Normalized).
Let S be an SPN and $\Sigma(S)$ be the set of all sum nodes of S. S is normalized iff, for all $\sigma \in \Sigma(S)$, $\sum_{c \in Ch(\sigma)} w_{\sigma c} = 1$ and $0 \leq w_{\sigma c} \leq 1$, where $w_{\sigma c}$ is the weight from edge $\sigma \rightarrow c$.

Definition 2.2 (Completeness).
Let S be an SPN and $\Sigma(S)$ be the set of all sum nodes of S. S is complete iff, for all $\sigma \in \Sigma(S)$, $Sc(i) = Sc(j), i \neq j; \forall i, j \in Ch(\sigma)$.

Definition 2.3 (Consistency)**.**
Let S be an SPN, $\Pi(S)$ be the set of all product nodes of S and X a variable in $Sc(S)$. S is consistent iff X takes the same value for all elements in $\Pi(S)$ that contain X.

Definition 2.4 (Validity)**.**
An SPN S is valid iff it always computes the correct probability of evidence S represents.

Theorem 2.1. *An SPN S is valid if it is both complete and consistent.*

Validity guarantees that the SPN will compute not only the correct probability of evidence, but also in time linear to its graph's edges. Therefore, it is preferable to learn valid SPNs. Notice that Theorem 2.1 is not restricted by completeness and consistency. In fact, incomplete and/or inconsistent SPNs can compute the probability of evidence correctly, but consistency and completeness guarantee that all sub-SPNs are also valid.

Definition 2.5 (Decomposability)**.**
Let S be an SPN and $\Pi(S)$ be the set of all product nodes in S. S is decomposable iff, for all $\pi \in \Pi(S)$, $Sc(i) \cap Sc(j) = \emptyset, i \neq j; \forall i, j \in Ch(\pi)$.

It is clear that decomposability implies consistency, therefore if an SPN is both complete and decomposable, than it is also valid. We choose to work with decomposability because it is easier to learn decomposable SPNs then it is to learn consistent ones. We do not lose representation power because a complete and consistent SPN can be transformed into a complete and decomposable SPN in no more than a polynomial number of edge and node additions [Peh+15]. We can now formally define an SPN.

Definition 2.6 (Sum-product network)**.**
A sum-product network (SPN) is a weakly connected DAG that can be recursively defined as follows.

An SPN:

*(1) with a single node is a univariate tractable probability distribution (**leaf**);*
*(2) is a normalized weighted sum of SPNs of same scope (**sum**);*
*(3) is a product of SPNs with disjoint scopes (**product**).*

The value of an SPN is defined by its type. Let λ, σ and π be a leaf, sum and product respectively. The values of such SPNs are given by $\lambda(\mathbf{x})$, $\sigma(\mathbf{x})$ and $\pi(\mathbf{x})$, where $\mathbf{x}$ is a certain evidence instantiation.

Leaf: *Let ϕ_λ be a univariate tractable distribution of variable X. $\lambda(\mathbf{x}) = \phi_\lambda(\mathbf{x})$.*
Product: $\pi(\mathbf{x}) = \prod_{c \in Ch(\pi)} c(\mathbf{x})$.
Sum: $\sigma(\mathbf{x}) = \sum_{c \in Ch(\sigma)} w_{\sigma c} c(\mathbf{x})$, *with* $\sum_{c \in Ch(\sigma)} w_{\sigma c} = 1$ *and* $0 \leq w_{\sigma c} \leq 1$.

Note that this definition assumes an SPN to be complete, decomposable and normalized. Other definitions in literature may differ from ours, but as we have mentioned before, for our implementation, this definition is convenient for us. Another observation worthy of notice is the value of $\lambda(\mathbf{x})$. Although here we consider $\mathbf{x}$ to be a multivariate instantiation (i.e. a set of — potentially multiple — variable valuations), we had initially defined a leaf to be a univariate distribution. Although it is possible to attribute leaves as multivariate probability distributions [RL14], for our definition we have chosen to keep a leaf's scope a unit set. Therefore, in the case of a leaf's value, $\mathbf{x}$ is a singleton (univariate) variable instantiation.

3. The `LearnGD` Schema

The `LearnGD` schema was proposed by Robert Gens and Pedro Domingos on *Learning the Structure of Sum-Product Networks* [GD13]. In this section we will outline the schema in pseudo-code and analyze a few properties derived from the algorithm.

Algorithm 1 `LearnGD`

```
Input  Set $\mathbf{D}$ of instances (data)
Input  Set $\mathbf{V}$ of variables (scope)
Output  An SPN representing a probability distribution given by $\mathbf{D}$ and $\mathbf{V}$
 1: if $|\mathbf{V}| = 1$ then                                  ▷ univariate data sample
 2:     return univariate distribution estimated from $D[\mathbf{V}]$ (data of $\mathbf{V}$)
 3: end if
 4: Take $\mathbf{V}$ and find mutually independent subsets $\mathbf{V}_i$ of variables
 5: if possible to partition then                 ▷ i.e. we have found independent subsets
 6:     return $\prod_i$ LearnGD ($\mathbf{D}$,$\mathbf{V}_i$)
 7: else                                            ▷ we cannot say there is independence
 8:     Take $\mathbf{D}$ and find $\mathbf{D}_j$ subsets of similar instances
 9:     if possible to partition then
10:         return $\sum_i \frac{|\mathbf{D}_j|}{|\mathbf{D}|}\cdot$ LearnGD ($\mathbf{D}_j$,$\mathbf{V}$)
11:     else                                              ▷ i.e. data is one big cluster
12:         return fully factorized distribution.
13:     end if
14: end if
```

Let us now, for a moment, suppose that SPNs are not necessarily complete, decomposable and normalized. We shall prove a few results derived from SPNs generated by Algorithm 1.

Lemma 3.1. *An SPN S generated by* `LearnGD` *is complete, decomposable and normalized.*

Proof. Lines 4–6 show that the scope of each child in a product node of S is a partition of the scope of their parent. Therefore, children have pairwise disjoint scopes on line 6, which proves decomposability for this part of the algorithm. In lines 8–10, since we are clustering similar instances, $\mathbf{D}$ is being partitioned but we are not changing $\mathbf{V}$ in any way. In fact, line 10 shows that we pass $\mathbf{V}$ to all other children. That is, all children of sum nodes have the same scope as their parent, which proves completeness. Let $\mathbf{D}_1, \ldots, \mathbf{D}_n$ be the subsets of similar instances. By the definition of clustering, $\mathbf{D}_1 \cup \ldots \cup \mathbf{D}_n = \mathbf{D}$ and $\mathbf{D}_i \cap \mathbf{D}_j = \emptyset$, $i \neq j$, $1 \leq i, j \leq n$. Thus it follows that $\sum_{i=1}^{n} \frac{|\mathbf{D}_i|}{|\mathbf{D}|} = 1$ and thus line 10 always creates complete and normalized sum nodes. Line 12 is a special case where, if we have discovered that $\mathbf{D}$ is one big data cluster, we shall create a product node π in which all children of π are leaves and

$$\bigcup_{\lambda \in \mathrm{Ch}(\pi)} \mathrm{Sc}(\lambda) = \mathrm{Sc}(\pi).$$

In other words, we fully factorize our product node into leaves. In this case, it is obvious that this product node is decomposable. □

`LearnGD` can be divided into four parts:

(1) Is the data univariate? If it is, return a leaf.
(2) Are partitions of the data independent? If they are, return a product node whose children are the independent partitions.
(3) Are partitions of the data similar? If they are, return a sum node whose children are the partition clusters.
(4) In case all else fails, we have a fully factorized distribution.

Going back to our definition of an SPN, we can now take a more intuitive approach and make the following observations:

(1) A leaf is nothing but a local/partitioned/sample distribution of a probability distribution given by a single variable.
(2) A product node determines independence between variables.
(3) A sum node is a clustering of similar data values (i.e. instances that are "alike").

This gives more semantic value to SPNs, whilst still retaining its expressivity. Following this approach, one can easily notice that each "layer" corresponds to a recursive call in `LearnGD`. In fact, each recursive call constructs a hidden layer that tries to partition the SPN even further. This gives SPNs a deep architecture that resembles deep models in that the deeper the model, the more representation power it has [DB11].

Let us now observe the scope of each type of node. A leaf is the trivial case, since it has a single variable in its scope by definition. Each layer above it can have either sum or product nodes. Let us now look at decomposability, that is: if a variable X appears in a child of a product node π, then X cannot appear in another child of π. This gives us the following result:

Lemma 3.2. *Let S be an SPN generated by `LearnGD`, and let $\Lambda(S)$ be the set of all leaves of S. Then, $\forall \lambda \in \Lambda(S)$, we have that, $\forall p \in Pa(\lambda)$, p is a product node.*

Proof. Our proof is by contradiction. Let us assume that $\exists p \in \mathrm{Pa}(\lambda)$ such that p is a sum node and $\exists c^* \in \mathrm{Ch}(p)$ a leaf. From our assumption that p is a sum node, we have that, since the SPN is complete, the scope of all children of p are the same and are all equal to the scope of p. Now let $c \in \mathrm{Ch}(p)$. There must exist another child c such that $c \neq c^*$ because of lines 5 and 9. From that we have $\mathrm{Sc}(c) = \mathrm{Sc}(c^*)$ because of completeness, and since $\mathrm{Sc}(c^*)$ is singular, then c must also be leaf. But it is impossible to have leaves with same scope and same parent (line 1 from Algorithm 1). Therefore, p is actually a product node. □

Lemma 3.3. *An SPN generated by* `LearnGD` *is a rooted tree.*

Proof. It suffices to show that for any vertex, its indegree is exactly one. We can prove that by saying that Algorithm 1 never adds edges between two already existing vertices. □

Definition 3.1. *A sum-product network that is a rooted tree is called a sum-product tree (SPT).*

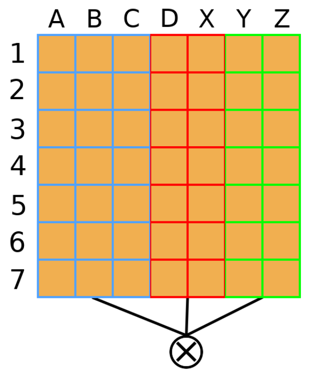
Theorem 3.1.
Let S be an SPT generated by `LearnGD`*. Let $n = |\,Sc(S)|$ and $m = |\mathbf{D}|$, where $\mathbf{D}$ is the data sent as parameter to* `LearnGD`*. Let h be the height of S. Then*

$$1 \leq h \leq n + m - 1$$

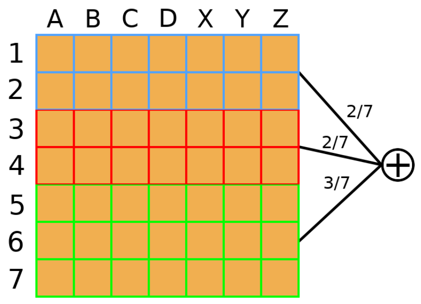
Proof. Sketch of proof: every sum or product node creates one more hidden layer (increments SPT's height by one) and decrements either an instance (if sum node) or variable in node's scope (if product node) by one at least. Last (deepest) sum node must have at least two product nodes as children (following Lemma 3.2), with each having one data instance each, therefore the number of layers created by sum nodes is at most $m - 1$. Similarly for product nodes, if we want to maximize the number of layers, the deepest product node must have two leaves as children, bringing the total count of product nodes to $n - 1$. Counting the last layer that is made out of leaves, we have $(n - 1) + (m - 1) + 1 = n + m - 1$. Furthermore, the SPT has the form of alternated sum-product layers (a sum node will follow a product node and vice-versa). This guarantees that each sum node modifies the overall data enough that the independence test will judge an independent variable from all others. The base case $h = 1$ is trivial: a size 1 scope generates one leaf with distribution equal to data. The case $h = 2$ is more interesting and occurs when all variables are dependent and belong to the same cluster. □

From Algorithm 1 we have learned that `LearnGD` can be structured into three parts. The first is discovering variable independencies and judging whether we should partition V and create a new product node. The second is, if the first part has failed, we must find possible clusters from the data we have. From these newly discovered clusters, we decide if we should create another sum node and assign each of its children a partition of these clusters, or if these clusters all form a single big all encompassing cluster. If this is the case, we create a new product node whose children are the fully factorized form of the present data. Finally, the third and last part is the base case. If the scope is of a single variable, we return the univariate probability distribution given by the univariate data.

If we were to visualize our dataset as a table where rows are instances and columns are variables, we could equate the algorithm as splitting, either horizontally or vertically, the dataset according to our partitioning decisions. For instance, if we had decided that there was a certain subset of variables that were independent of the rest of the variables, we would "split" our dataset table vertically, with each subtable belonging to a variable subset. Similarly for cluster partitioning, we would split our dataset table horizontally. Figure 1 illustrates the procedures for variable splitting (Figure 1a) and instance splitting (Figure 1b).



(A)



(B)

FIGURE 1. These two images represent a dataset in table form. Rows are instances and columns are variables.Figure 1a shows variable splitting. In this example, we have observed that the subsets $\mathbf{V}_1 = \{A, B, C\}$, $\mathbf{V}_2 = \{D, X\}$ and $\mathbf{V}_3 = \{Y, Z\}$ are independent of each other. That is, for every pair $(P, Q), P \in \mathbf{V}_i, Q \in \mathbf{V}_j, i \neq j$, P is independent of variable Q. Given these partitions, we create a new product node whose children are the recursive calls to `LearnGD`. Note that their new scopes are now V_i, and their data instance covers only their new scope. In this example, the new partitions form subtables whose columns are adjacent to each other (e.g. A, B and C are adjacent). However, we could find an independent subset that did not necessarily obey a graphical rule, that is, we could have found that A and Y belong to the same partition. In this case, we would have considered A and Y as a new subset, regardless of their graphical positions. For Figure 1b, we apply a similar concept. In this case we are clustering similar instances. Note that instance partitioning does not alter their scope. Each instance subset $\mathbf{D}_1 = \{1, 2\}$, $\mathbf{D}_2 = \{3, 4\}$ and $\mathbf{D}_3 = \{5, 6, 7\}$ equates to a discovered cluster. A new sum node is then created, with weights corresponding to the ratio of rows in each subtable. The subtables are then added as children of the sum node and then recursed. Just like with variable splitting, these partitions do not necessarily obey a graphical rule. We could have non-adjacent rows as a single partition.

Now that we have the general idea of the algorithm, we shall describe and analyze how to do both variable and instance splitting. We will reserve a section to each of these topics. Once we have covered them both, we shall once again take a broader look at the `LearnGD` schema and work on some other results that depend on the two next sections.

4. Variable Independence

The core of variable splitting is finding independence between variables. What we wish to find is partitions of the current SPN scope such that every element in a partition is independent of all other elements in other partitions. In this section we shall explain the general idea, describe and analyze the method used in our implementation and lastly discuss certain problems encountered during experiments and implementation.

The description of our problem is: given a dataset with a set of variables $\mathbf{V}$, we wish to find a set $\mathbf{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n\}$, where $\mathbf{P}_i \cap \mathbf{P}_j = \emptyset, i \neq j$ and $\mathbf{P}_1 \cup \ldots \cup \mathbf{P}_n = \mathbf{V}$ and $\mathbf{P}_i$ is a subset of variables. That is, a set of partitions of $\mathbf{V}$. Additionally, for every $\mathbf{P}_i$ and $\mathbf{P}_j$, $i \neq j$, $\forall u \in \mathbf{P}_i, v \in \mathbf{P}_j$, $u \perp v$ (u is independent of v), where $\perp$ is the independence operator.

Suppose we have an independence oracle Ω that tells us if $X \perp Y$ and does so in constant time. A naïve solution to this problem is, for every pair $X, Y \in \mathbf{V}$, ask Ω if $X \perp Y$. We then memorize which ones are dependent and which are independent. For such memoization, we can use an undirected graph.

Definition 4.1 (Independence Graph).
Let $G = (\mathbf{V}, \mathbf{E})$ be an undirected graph with vertex set $\mathbf{V}$ and edge set $\mathbf{E}$. Let i and j be vertices in $\mathbf{V}$. There exists an edge e_{ij} iff $i \not\perp j$.

This reduces our problem to one of finding connected subgraphs. Since there exists an edge if and only if the two connected variables are dependent, to find a partition $\mathbf{P}_i$, it suffices to find a connected subgraph in which all of its vertices have no path to another subgraph (there is no dependence path between variables).

Proposition 4.1. *If $H = (\mathbf{V}', \mathbf{E}')$ is a connected subgraph in Independence Graph $G = (\mathbf{V}, \mathbf{E})$ then $\mathbf{V}' \in \mathbf{P}$.*

Proof. Since there cannot be two same variables in our graph, it suffices to show that, $\forall u' \in \mathbf{V}', u \in \mathbf{V} \setminus \mathbf{V}'$: $u' \perp u$. Let us assume that $\mathbf{V}' \notin \mathbf{P}$. That is, there exists an element in $\mathbf{V}'$ that is dependent of an element in $\mathbf{V} \setminus \mathbf{V}'$. That means there exists a vertex $v' \in \mathbf{V}'$ and another vertex $v \in \mathbf{V} \setminus \mathbf{V}'$ such that an edge $e_{v'v}$ connects both of them. But $u \in \mathbf{V} \setminus \mathbf{V}'$ and $\mathbf{V}'$ is the vertex set of a connected subgraph, which means such an edge cannot exist, as there is no path from a vertex of H to a vertex in $\mathbf{V} \setminus \mathbf{V}'$. Therefore our assumption that $\mathbf{V}' \notin \mathbf{P}$ is false. □

Sketching our naïve solution, we have:

Algorithm 2 `IndepGraph`

```
Input  Set $\mathbf{D}$ of instances (data)
Input  Set $\mathbf{V}$ of variables (scope)
Output  A set $\mathbf{P}$ of independent partitions of variables
 1: Let $\Omega$ be an independence oracle that returns true if independent and false if
    dependent
 2: Let $G = (\mathbf{V}, \mathbf{E} = \emptyset)$ be an empty independence graph
 3: for each variable $X \in \mathbf{V}$ do
 4:     for each variable $Y \neq X, Y \in \mathbf{V}$ do
 5:         if $\Omega(X, Y) =$ false then
 6:             Let $e_{XY}$ be a new edge connecting $X$ and $Y$
 7:             $\mathbf{E} \leftarrow \mathbf{E} \cup e_{XY}$
 8:         end if
 9:     end for
10: end for
11: $\mathbf{P} \leftarrow$FindConnectedSubgraphs(G)
12: return $\mathbf{P}$
```

We can find the connected subgraphs using a Union-Find structure. Since Union-Find is out of the scope of this paper, we shall not go into a deep discussion of it. However, we shall assume we have a Union-Find implementation which uses both *union by rank* and *path compression* heuristics, bringing the complexity of a series of m Union-Find operations to have amortized time $\mathcal{O}(m \log_2^*(n))$, where n is the number of elements and $\log^*$ is the iterated logarithm function.

Algorithm 3 `FindConnectedSubgraphs`

```
Input  Graph $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is vertex set and $\mathbf{E}$ is edge set
Output  A set $\mathbf{S}$ of the sets of vertices of the connected subgraphs of $G$
 1: Let $\mathbf{U}$ be the set of Union-Find structures.
 2: for each variable $X \in \mathbf{V}$ do
 3:     Let $u$ be a new Union-Find structure whose representative is $X$.
 4:     $u \leftarrow$ MakeSet($X$)
 5:     $\mathbf{U} \leftarrow \mathbf{U} \cup u$
 6: end for
 7: for each variable $X \in \mathbf{V}$ do
 8:     for each variable $Y \in \mathbf{V}$ do
 9:         $u_X \leftarrow$Find($X$)
10:         $u_Y \leftarrow$Find($Y$)
11:         if $u_X \neq u_Y$ then Union($u_X$, $u_Y$)
12:         end if
13:     end for
14: end for
15: Let Convert($\cdot$) be function that converts Union-Finds to set of sets.
16: return Convert($\mathbf{U}$)
```

Let $n = |\mathbf{V}|$, the number of variables. The number of elements in all Union-Finds is n, as we can from lines 2–6. We also know from lines 7–14 that the number of Union-Find operations we call is $(2n)^2 + n(n-1) = 5n^2 - n$. Substituting these values into the amortized complexity of our Union-Find implementation gives

$$(5n^2 - n)\log_2^* n.$$

Since we are always differentiating edges $X - Y$ and $Y - X$, we can slightly improve performance by only taking into account one of those edges. We can decrease our number of Union-Find operations to $\binom{n}{2}$, which is the number of combinations if we choose 2 in n. The final complexity comes down to

$$\binom{n}{2}\log_2^* n = \left(\frac{n!}{(n-2)!2!}\right)\log_2^* n = \left(\frac{n(n-1)}{2}\right)\log_2^* n$$

Let $\alpha(n) = \log_2^* n$. We know that $\alpha(n)$ grows extremely slow, therefore we will assume $\alpha(n)$ as a constant, giving the final amortized complexity of Algorithm 3 the asymptotic form of $\mathcal{O}((n^2/2 - n/2)\log_2^* n) = \mathcal{O}(n^2\alpha(n) - n\alpha(n)) = \mathcal{O}(n^2)$.

Before we analyze `IndepGraph`, we need to review our previous assumption that a certain independence oracle Ω returns, in $\mathcal{O}(1)$, whether two variables are independent of each other given data. We will now show the independence test we used for our implementation. We have implemented two independence tests, however we shall only describe one in our analysis. The first is the standard Pearson's chi-square independence test and the second is the G-test. Both of these tests use the chi-square distribution. In our implementation we provide two options for computing the cumulative probability function (CDF) of a chi-square distribution of k degrees of freedom. One is based on the GNU Scientific Library (GSL) written in C, and the other is a Go implementation that calls Go's gamma function implementation. To simplify our analysis, we assume both of these implementations have same execution time $T(n)$. Furthermore, it is widely known that Pearson's and G-test's have same asymptotic time. In this paper we will assume all independence tests were run using the G-test. Therefore, when we refer to an "independence test" or "oracle", we mean we use a G-test to infer the dependency relation between two variables.

We shall call our independence test an `Oracle`. Our implementation is defined in pseudo-code as Algorithm 4. Let X be a variable and $\mathbf{D}$ be the dataset. We shall call $\text{Val}(X)$ the possible valuations of X. That is, if we had a variable *Animal* and we were only considering cats and dogs, we would then say that the possible valuations of *Animal* is defined by the set $\text{Val}(\mathit{Animal}) = \{\mathit{Cat}, \mathit{Dog}\}$. Let $\mathbf{V}$ be the set of variables in our local scope, and $\mathbf{v}$ be the set of instantiations of $\mathbf{V}$. Let us denote the cartesian product $\underbrace{\mathbf{v} \times \cdots \times \mathbf{v}}_{n}$ as $\mathbf{v}^n$, and $N : \mathbf{v}^n \to \mathbb{Z}^*$, where N is a function that takes a set of valuations $\mathbf{v}$ and returns the number of instantiations that are consistent with $\mathbf{v}$ in $\mathbf{D}$. For instance, take our previous example where we have two possible valuations for *Animal*: *Cat* and *Dog*. Suppose our D shows we have three cats and four dogs in our data. Then $N[\mathit{Animal} = \mathit{Cat}] = 3$ and $N[\mathit{Animal} = \mathit{Dog}] = 4$. If we had a second variable *Behavior* with $\text{Val}(\mathit{Behavior}) = \{\mathit{Agitated}, \mathit{Calm}\}$, and D showed:

Animal	**Behavior**
Dog	Agitated
Cat	Calm
Cat	Agitated
Dog	Agitated
Cat	Calm
Dog	Calm
Dog	Agitated

We could say that $N[Animal = Cat, Behavior = Calm] = 2/3$. Algorithm 4 builds a contingency table from data D, computing the N counts of each possible instantiations of the two populations. We call these counts the observed frequencies. We then compute the expected frequencies from the ratios of the totals. Next we compute $G = 2\sum_i O_i \cdot \ln\left(\frac{O_i}{E_i}\right)$, find the CDF of G in the corresponding chi-square distribution and compare with a certain significance value. If the area found on the chi-square distribution is less than the significance value, than we can, under the null hypothesis, reject the fact that they are independent.

Let A be an $m\times n$ matrix with elements denoted by a_{ij}. We shall use the notation $A[i:j][p:q]$ as the submatrix derived from taking the columns $[a_{ip}, \ldots, a_{iq}]$, …, $[a_{jp} \ldots, a_{jq}]$ from A. To simplify our algorithm, we assume $\text{Val}(X)$ is an ordered set and is indexed from 1.

Algorithm 4 `Oracle`

```
Input  Variables $X$ and $Y$
Input  Dataset $D$
Output  Returns the result of the evaluation $X \perp Y$
 1: Let $p = |\text{Val}(X)|$ and $q = |\text{Val}(Y)|$
 2: Let $C$ be a $(p+1)\times(q+1)$ matrix                ▷ $C$ is the contingency table
 3: Let $C^*$ be the submatrix $C[1:p][1:q]$
 4: for $i \leftarrow 1$ to $p$ do
 5:     $\nu_X \leftarrow \text{Val}(X)[i]$
 6:     for $j \leftarrow 1$ to $q$ do
 7:         $\nu_Y \leftarrow \text{Val}(Y)[j]$
 8:         $c_{ij} \leftarrow N[X = \nu_X, Y = \nu_Y]$
 9:         $c_{(p+1,j)} \leftarrow N[Y = \nu_Y]$
10:     end for
11:     $c_{(i,q+1)} \leftarrow N[X = \nu_X]$
12: end for
13: $c_{(p+1,q+1)} \leftarrow N[X = \cdot, Y = \cdot]$          ▷ Total number of instances
14: $E_{ij} = \frac{c_{(p+1,j)}c_{(i,q+1)}}{c_{(p+1,q+1)}}$, for $i = 1,\ldots,p$ and $j = 1,\ldots,q$
15: $g \leftarrow \sum_{i=1}^{p}\sum_{j=1}^{q} c_{ij} * \ln\left(\frac{c_{ij}}{E_{ij}}\right)$
16: Let $F$ be the CDF for $\chi^2((p-1)\cdot(q-1))$, and $\sigma$ be the significance value
17: return $F(g) \geq \sigma$
```

Algorithm 4 has time $\mathcal{O}(|Val(X)| \cdot |Val(Y)| + T(m))$. If we consider a series of `Oracle` operations on all variables, we can conclude that our worst time asymptotically would be $\mathcal{O}(m^2 + T(m))$, where $m = \max |Val(V)|$. Going back to Algorithm 2, we now have that our independence oracle no longer takes time $\mathcal{O}(1)$. In fact, our `IndepGraph` routine has time

$$n^2 \cdot (m^2 T(m)) + \frac{n(n-1)}{2} \log_2^* n$$

where $n = |\mathbf{V}|$. Let us assume, for simplicity, that $\mathcal{O}(T(x)) = \mathcal{O}(1)$. The asymptotic time is then

$$\mathcal{O}\left(n^2 m^2 + \frac{n(n-1)}{2} \log_2^* n\right) = \mathcal{O}(n^2 m^2 + n^2) = \mathcal{O}(n^2 m^2)$$

Our implementation is extremely slow when n and m are moderately large. On some experiments we shall mention later, we dealt with $n = 2576$ and $m = 8$. Running `IndepGraph` became impractical. We can improve the runtime of `IndepGraph` using a heuristic based on Union-Find. The idea of the heuristic is to avoid evaluating edges that do not change the complete subgraphs in representability.

Definition 4.2 (Minimal Independence Graph). *Let $I = (\mathbf{V}, \mathbf{E})$ be an independence graph and $S = (\mathbf{V}, \mathbf{E}')$ an undirected graph. Let $\mathbf{H}$ be the set of connected subgraphs in I and $\mathbf{Z}$ be the set of connected subgraphs of S. S is a minimal (independence graph) of I if for every $H \in \mathbf{H}$ with vertex set $\mathbf{V}_H$, there exists one and only one $Z \in \mathbf{Z}$, and Z is a spanning tree of H.*

Definition 4.3 (Representability). *Let $G = (\mathbf{V}, \mathbf{E})$ and $H = (\mathbf{V}, \mathbf{E}')$ be undirected graphs. Let $R(X, Y, G)$ be a function that takes two variables X and Y, and an undirected graph G and returns whether X is reachable from Y. G is as representable as H if, for every pair of vertices $X, Y \in \mathbf{V}$, $R(X, Y, G) = R(X, Y, H)$.*

Theorem 4.1. *Let I be an independence graph and S be its minimal independence graph. I is as representable as S.*

Proof. Let $\mathbf{H}$ and $\mathbf{Z}$ be the set of connected subgraphs of I and S respectively. To prove I is as representable as S we must prove that, for every $H \in \mathbf{H}$ and its equivalent spanning tree $Z \in \mathbf{Z}$, $R(X, Y, H) = R(X, Y, Z)$. This condition suffices because the case of X and Y not being in the same connected subgraph is already defined as non-reachable from the definition of connected subgraphs. Since Z is a spanning tree of H, from the definition of spanning tree, we know that there exists a path from X to Y if they are both in Z, which is also true in the case of H. If X is not reachable from Y, then X and Y are in different connected subgraphs, in which case $R(X, Y, H) = R(X, Y, Z) =$ `false`. □

Definition 4.4 (Irrelevant Edge). *Let $I = (\mathbf{V}, \mathbf{E})$ be an independence graph and $S = (\mathbf{V}, \mathbf{E}')$ be a minimal independence graph of I. An edge $e_{ij} \in \mathbf{E}$ connecting vertices $i, j \in \mathbf{V}$ is an irrelevant edge wrt S if $e_{ij} \notin \mathbf{E}'$.*

Lemma 4.1. *Let $I = (\mathbf{V}, \mathbf{E})$ be an independence graph and $G = (\mathbf{V}, \mathbf{E}')$ an undirected graph where $\mathbf{E}' \subseteq \mathbf{E}$. G is minimal wrt I iff there are no irrelevant edges in $\mathbf{E}'$.*

Proof. Follows immediately from the definition of irrelevant edges and minimal independence graph. □

We want to find minimal independence graphs, instead of a full independence graph. Algorithm 5 finds a minimal independence graph by using Union-Find.

Algorithm 5 `IndepGraphUF`

```
Input  Set $\mathbf{D}$ of instances (data)
Input  Set $\mathbf{V}$ of variables (scope)
Output  A set of independent partitions of variables
1: Let $\Omega$ = Oracle that returns true if independent and false if dependent
2: Let $G = (\mathbf{V}, \mathbf{E} = \emptyset)$ be an empty undirected graph
3: Let $\mathbf{U}$ be a new empty set of Union-Find structures
4: for each variable $X \in \mathbf{V}$ do
5:     Let $u$ be a new Union-Find structure whose representative is $X$
6:     $u \leftarrow$ MakeSet $(X)$
7:     $\mathbf{U} \leftarrow \mathbf{U} \cup u$
8: end for
9: ▷ Invariant 1
10: for each variable $X \in \mathbf{V}$ do
11:     for each variable $Y \in \mathbf{V}$ do
12:         $u_X \leftarrow$ Find $(X)$
13:         $u_Y \leftarrow$ Find $(Y)$
14:         if $u_X \neq u_Y$ then
15:             ▷ Invariant 2
16:             if $\Omega(X, Y, \mathbf{D})$ = false then
17:                 Let $e_{XY}$ be a new edge connecting $X$ and $Y$
18:                 $\mathbf{E} \leftarrow \mathbf{E} \cup e_{XY}$
19:                 Union $(u_X, u_Y)$
20:                 ▷ Invariant 3
21:             end if
22:         end if
23:         ▷ Invariant 4
24:     end for
25: end for
26: Let Convert $(\cdot)$ be a function that converts Union-Finds to set of sets.
27: return Convert $(\mathbf{U})$
```

Lemma 4.2. *Let I be an independence graph and S an undirected graph generated by `IndepGraphUF`. S has no irrelevant edges.*

Proof. Let us first list all the invariants of Algorithm 5:

Invariant 1: each variable $X \in \mathbf{V}$ is in a disjoint set wrt all other variables.
Invariant 2: X is in a different set than Y.
Invariant 3: e_{XY} is not an irrelevant edge and X and Y are in the same set.
Invariant 4: All edges connecting X to another vertex have already been created.

When we reach Invariant 1, we know that every vertex is in its own disjoint set, which is equivalent to our graph G whose edge set is empty. Once we iterate through each variable, we verify whether the pair of variables $X, Y \in \mathbf{V}$ are in the same set. If they are, then it means we there already exists a path X to Y in our graph G, and thus e_{XY} would be an irrelevant edge. If they are not in the same set, then it means there exists no path from X to Y, which means we must verify if $X \perp Y$. Invariant 3 shows that, if we are to add such an edge e_{XY}, then X and Y were not reachable previous to adding the edge. Furthermore, we guarantee that X and Y are now in the same set and thus in the same connected subgraph. Once we reach Invariant 4, we know that we have attempted all possible edges e_{XY} such that X and Y are in different connected subgraphs. We ignore adding edges that connect two vertices that are in the same connected subgraph. This guarantees that we add no irrelevant edges. □

Theorem 4.2. *If I is an independence graph and S is an undirected graph generated by* `IndepGraphUF`*, then S is minimal wrt I.*

Proof. Follows immediately from Lemma 4.2 and Lemma 4.1. □

Let $n = |\mathbf{V}|$ and $m = \max|\operatorname{Val}(X)|, \forall X \in \mathbf{V}$, and assuming function `Convert` has asymptotic time equal to $\mathcal{O}(n)$, then the worst case for Algorithm 5 is the same for Algorithm 2: when all variables are in their own disjoint sets and there are no edges in G. When that happens, we have

$$\mathcal{O}(n + n^2 \cdot m^2 + n) = \mathcal{O}(n^2 m^2).$$

Which is the same time for `IndepGraph`. However, the Union-Find heuristic decreases time significantly in the general case. In fact, if we analyse the best case for `IndepGraph`, we find that the complexity for such instance is $\Omega(n^2m^2)$, which in turn gives us a tight bound $\Theta(n^2m^2)$ on our algorithm with no heuristics. If we examine `IndepGraphUF` however, we find that our best case is much faster and occurs when our graph is fully connected.

Consider the graph G in Algorithm 5. At the point of Invariant 1, G has no edges and thus all vertices are disjoint sets. At each iteration of vertices X, Y such that $X \neq Y$, we check if they are in the same set. Assuming `Find` to compute the set representative in constant time, we can find whether the two vertices are in the same set in time $\mathcal{O}(1)$. On the first iteration we have, from Invariant 1, that they are not in the same set, possibly causing a new edge to be created. This takes time $\mathcal{O}(m^2)$ because of `Oracle` (note we are assuming `Union` to run in constant time). Next we decide, from the result of `Oracle`, whether we add a new edge or not. If we decide not to add a new edge, we will eventually have to pass through the two vertices multiple more times in order to test their independency with other vertices. If we decide to add a new edge e_{XY}, we will only look to one of them instead (their representative). Therefore we shall consider e_{XY} to have been created. The next step checks another pair of vertices. The best case occurs when the pair of vertices being tested are X, Z, where X is one of the vertices that have already been already tested and added to the set. Under the same hypothesis as with the previous pair, we add a new edge e_{XZ} and add Z to the set of X. This gives us a cost of $\mathcal{O}(m^2)$. If we keep this same routine with all vertices in G, we have that the final cost of all operations is the sum of each of these $\mathcal{O}(m^2)$ costs, which means our best case is given by $\Omega(n \cdot m^2)$, which is better than `IndepGraph`.

Since finding a minimal independence graph is sufficient to our uses, we shall refer to an independence graph as one of its minimal. For this reason, we shall always use `IndepGraphUF` instead of `indepGraph`. A brief analysis of the memory usage in `IndepGraphUF` gives us a linear amount of memory used. Suppose each vertex uses a single unit of memory. We never create more units except when we create a set of Union-Find for each vertex. This gives us $2n$ units of memory. If we count the units used in `Oracle`, we then have $2n + (m+1)^2$. Therefore, our memory usage's upper bound is $\mathcal{O}(2n + (m+1)^2) = \mathcal{O}(2n + m^2)$.

5. Clustering

The task of clustering is associated with the function of the sum node. We wish to classify subsets of instances that present similarities. In this section we present two types of clustering we implemented. The first is k-means clustering, a simple partitioning algorithm that uses the centroid of a fixed number of k clusters to identify in which partition each instance belongs to. The second is DBSCAN, a density algorithm that automatically identifies the number of possible clusters based on the distance between high-density groups of instances.

Algorithm 6 `KMeans`

```
Input An integer $k$ that represents the fixed number of clusters to be found
Input Set $\mathbf{D}$ of elements
Output A set of partitions $\mathbf{P}$, with each partition representing a cluster
 1: Let $\mathbf{P} = \{P_1, P_2, \dots, P_k\}$ be the set of partitions
 2: Let $\mathbf{c} = \{c_1, c_2, \dots, c_k\}$ be each centroid where $c_i$ represents $P_i$
 3: $c_1 \leftarrow 0, c_2 \leftarrow 0, \dots, c_k \leftarrow 0$
 4: for each $c_i$ do
 5:     Select an element $e_j \in \mathbf{D}$ that has not been previously chosen
 6:     $c_i \leftarrow e_j$
 7: end for
 8: repeat
 9:     for each $e_i \in \mathbf{D}$ do
10:         Let $d(e_i, c_j)$ be the euclidean distance between $e_i$ and $c_j$
11:         Find $\arg\min_j d(e_i, c_j)$
12:         If $e_i$ belongs to a partition $P_k$, remove $e_i$ from $P_k$
13:         Add $e_i$ to $P_j$
14:     end for
15:     for each $c_i \in \mathbf{c}$ do
16:         Recompute mean of all $e_j \in P_i$
17:     end for
18: until convergence
19: return $\mathbf{P}$
```

K-means clustering is a simple clustering algorithm that finds a fixed number of k partitions of the set by iteratively calculating the centroid and measuring the distances between each element and each centroid. The algorithm starts with the k centroids equal to k element positions chosen randomly. At each step, for each element e in the set, we compute the k distances between e and a c_i centroid. We choose the centroid that minimizes such distance and then add e to the partition whose representative is c_i. We then recompute all centroids as the mean of each element in its partition. The algorithm stop criterion is when all centroids' last value is equal to their current value (i.e. there is no change in the centroids). We use the Forgy method [For65] for initializing the centroids.

The k-means clustering algorithm is known to have complexity $\mathcal{O}(mkdi)$, where m is the number of elements of $\mathbf{D}$, k is the number of clusters, d is the dimension of the elements, and i is the number of iterations until convergence. We know that d is the number of variables to be evaluated in the sum node. This gives us an upperbound of d being at most $|\mathbf{V}|$. This gives us a complexity of $\big(tk|\mathbf{D}||\mathbf{V}|\big)$, where t is an upperbound to i.

We can improve the runtime of our k-means implementation by recomputing the means only when we find that a new element has moved to a different partition. This improvement does not change the asymptotic complexity, however.

Although the k-means clustering algorithm is relatively fast in practice, it is restricted by the fixed number of clusters. We now show our implementation of the DBSCAN clustering algorithm. DBSCAN is a density based clustering algorithm, which means it searches for clusters based on the number of elements in a certain area. If there are enough many points in this area, we consider adding such elements to a cluster. Intuitively, we are searching for "element-dense" areas. We next formalize the notion of density based on the Ester *et al* article [Est+96].

Definition 5.1 (ε-neighborhood)**.** *Let $e_1 \in \mathbf{D}$ be an element and $d : \mathbf{D} \times \mathbf{D} \to \mathbb{R}$ the euclidean distance between two elements. The ε-neighborhood of e_1, denoted by $N_\varepsilon(e_1)$ is defined as $N_\varepsilon(e_1) = \{e_2 \in \mathbf{D} | d(e_1, e_2) \leq \varepsilon\}$.*

The ε-neighborhood of an element e is simply the set of all elements different than e that are at least at ε of distance from e. We refer to *core element* as the elements that are inside a cluster. *Border elements* are elements that are on the border of a cluster. Note that border elements are still in their respective cluster.

Definition 5.2 (Directly density-reachable)**.** *An element e_1 is directly density-reachable from another element e_2 with parameters ε and μ if $e_1 \in N_\varepsilon(e_2)$ and $|N_\varepsilon(e_2)| \geq \mu$.*

The parameters ε and μ define whether two points are close enough to be considered part of a cluster. Parameter ε defines the "area" to be examined and μ defines the "minimum density" or minimum number of elements for a cluster to be considered. The condition is assymetrical. The assymetric case guarantees that one of the elements is a border element.

Definition 5.3 (Density-reachable). *An element e_1 is density-reachable from another element e_k if there exists a chain of elements $e_1, \dots, e_k$ such that e_{i+1} directly density-reachable from e_i*

Definition 5.4 (Density-connected). *An element e_1 is density-connected to another element e_2 given parameters μ and ε if there exists another element e' such that e_1 is density-reachable from e' and e_2 is also density-reachable from e' wrt μ and ε.*

In other words, if two elements are density-connected, there exists a certain "path" between the two elements. Note that although such a path exists, it is not necessarily true that the two elements are density-reachable from each other, since density-reachability is not symmetric.

Definition 5.5 (Cluster). *Let $\mathbf{D}$ be a set of elements. Given parameters μ and ε, a cluster $\mathbf{C}$ is a subset of $\mathbf{D}$ such that:*

i (Maximality) Let e_1 and e_2 be elements in $\mathbf{D}$. If $e_1 \in \mathbf{C}$ and e_2 is density-reachable from e_2, then $e_2 \in \mathbf{C}$.
ii (Connectivity) Let e_1 and e_2 be elements in $\mathbf{C}$. Then e_1 is density-connected to e_2.

Additionally, for any pair of clusters $\mathbf{C}_i$ and $\mathbf{C}_j$: $\mathbf{C}_i \cap \mathbf{C}_j = \emptyset$.

Definition 5.6 (Noise). *Let $\mathbf{C_1}, \dots, \mathbf{C_m}$ be the set of clusters that form the set of elements $\mathbf{D}$ given parameters μ and ε. Noise is the set of elements $\mathbf{N}$ in $\mathbf{D}$ such that $\mathbf{N} = \{\forall e \in \mathbf{N} | \forall i : e \notin \mathbf{C}_i\}$.*

The DBSCAN algorithm works as follows. We choose an arbitrary element e and find all of density-reachable neighbours that obey the parameters μ and ε. If e is a core element, then construct a new cluster that contains e. Else, if it is a border element, choose another distinct element e' and retry. We iteratively pass through each element and apply the previous conditions. If we find two clusters that are close enough given μ and ε, we merge them into one. Our implementation uses a Union-Find structure as clusters. We use `Union` to merge two clusters together and `Find` to discover whether two elements are in the same cluster. Furthermore, we consider that each element has its own unique set at initialization. We shall use an $n \times n$ distance matrix M. A distance matrix is a matrix that stores the distances of each pair e_i and e_j of elements. That is, $M = (m_{ij})$ with $m_{ij} = d(e_i, e_j), 1 \leq i, j \leq n$, where $d(e_i, e_j)$ is the euclidean distance between elements e_i and e_j.

Algorithm 7 `DBSCAN`

```
Input Parameter $\mu$ that represents the minimum number of points
Input Parameter $\varepsilon$ that represents the minimum distance
Input Set $\mathbf{D}$ of elements
Output A set of partitions $\mathbf{P}$, with each partition representing a cluster
 1: Let $M$ be the distance matrix of $\mathbf{D}$
 2: Let $Q$ be a queue of elements
 3: Let $e$ be an arbitrary element $e \in \mathbf{D}$, Enqueue $(Q, e)$
 4: repeat
 5:     $e_i \leftarrow$Dequeue $(Q)$
 6:     Let $N_\varepsilon(e_i)$ be the $\varepsilon$-neighborhood of $e_i$
 7:     for each $e_j \in \mathbf{D}$, such that $e_i \neq e_j$ do
 8:         if $m_{ij} \leq \varepsilon$ and Find $(e_i) \neq$ Find $(e_j)$ then
 9:             $N_\varepsilon(e_i) \leftarrow N_\varepsilon(e_i) \cup e_j$
10:         end if
11:     end for
12:     if $N_\varepsilon(e_i) \geq \mu$ then
13:         for each $e_j \in N_\varepsilon(e_i)$ do
14:             Union $(e_i,\, e_j)$
15:             $e_i$.visited $\leftarrow$ true, $e_j$.visited $\leftarrow$ true
16:         end for
17:     end if
18:     if $Q$ is empty then
19:         Try to find an element $e_k$ such that $e_k$.visited = false
20:         if $e_k$ exists then
21:             Enqueue $(Q,\, e_k)$
22:         end if
23:     end if
24: until $Q$ is empty
25: return the converted set of Union-Find structures into a set of sets.
```

The worst case input for algorithm `DBSCAN` is when, for every element in $\mathbf{D}$, this element is not in a cluster given parameters μ and ε. Lines 7–11 in Algorithm 7 show the algorithm finding the ε-neighborhood of an arbitrary e_i element in $\mathbf{D}$. Finding such a neighborhood is $\mathcal{O}(n)$, where $n = |\mathbf{D}|$ as we must search for every other element not in the same cluster as e_i. We improve our running time by pre-computing a distance matrix as to avoid repetitive calculations when finding the ε-neighborhood. Pre-computing M is an $\mathcal{O}(n^2)$ task, as we must compute each distance for each pair. Next we find whether such a neighborhood is sufficiently dense. If it is, for each element in N_ε, we `Union` the two sets. Assuming $\mathcal{O}(1)$ on `Union` and `Find`, we have a complexity of $\mathcal{O}(n)$ in the worst case — when all elements are inside e_i's neighborhood. Finally, we find the next unvisited element, which is clearly found in $\mathcal{O}(n)$ time. The overall complexity of `DBSCAN` is $\mathcal{O}(n^2)$, with memory usage $\mathcal{O}(n^2)$ because of M.

As we shall see in the experiments section, although `DBSCAN` has complexity $\mathcal{O}(n^2)$, it is `IndepGraphUF` that takes most of the algorithm's run time.

6. Inference

Let S be an SPN. The probability of evidence of a set of events $\mathbf{X}$ is the value of the SPN divided by the partition function. The partition function of an SPN is the value of an SPN when all variables are unknown. We shall denote the partition function of S as $S(\cdot)$. Therefore, the value of the SPN is $S(\mathbf{X})/S(\cdot)$. When an SPN is normalized, the partition function is $S(\cdot) = 1$ and thus the probability of evidence is the value of the SPN itself. Since we have defined SPNs as normalized, we shall always consider $S(\cdot) = 1$.

The probability of evidence is computed as follows. Let r be the root node of S. The value of $S(\mathbf{X})$ is the value of the root node r given evidence $\mathbf{X}$. The value of a node is described in Definition 2.6. We recursively compute the value of each node given $\mathbf{X}$. For every product node π, the evidence is only relevant when $\text{Sc}(\pi) \cap \mathbf{X} \neq \emptyset$, since the children of product nodes have disjoint scopes.

Computing the probability of evidence requires only that we pass each node once to get the value of its children, and another time to compute the true value given its children. Therefore, we can achieve inference in time linear to the SPN's edges.

In order to avoid numerical errors, we use the logarithm of the values of each node instead of the true values. Therefore, the logarithm values of each node type is as follows:

Leaf:

Let λ be a leaf node and $\{X\} = \text{Sc}(\lambda)$. The value of the leaf node is

$$\lambda(\mathbf{X}) = \begin{cases} \ln(\lambda(X = x)) & \text{if } X \in \mathbf{X} \\ 0 & \text{else} \end{cases}$$

Sum:

Let σ be a sum node and $\mathbf{W} = \{w_1, \ldots, w_n\}$ be the weights of the n children of σ. The value of a sum node is the weighted sum of each of its children. Let $\mathbf{A} = \{a_1, \ldots, a_n\}$ a set where $a_i = \ln(w_i) + c_i(\mathbf{X})$ with $c_i \in \text{Ch}(\sigma)$. We shall reorder A such that $a_1 > \cdots > a_n$.

$$\sigma(\mathbf{X}) = a_1 + \ln\left(1 + \sum_{i=2}^{n} e^{(a_i - a_1)}\right)$$

Product:

Let π be a product node. The value of a product node is

$$\pi(\mathbf{X}) = \sum_{i=1}^{n} c_i(\mathbf{X}), \text{ where } c_i \in \text{Ch}(\pi).$$

For MAP, we slightly alter the SPN in order to maximize each children's value. We replace all sum nodes with max nodes. That is, the value of the sum node is now the value of the weighted child that has the maximum value. Since the children of sum nodes can be seen as classifications of categories of similar instances, we can see the max node as choosing the sub-SPN that has the most probability of appearing. That is, we choose the cluster that is most frequent. Let $\{X\} = \text{Sc}(\lambda)$, where λ is a leaf node. To maximize the value of λ we check whether $X \in \mathbf{X}$. If it is, then we simply return the log probability of X. Otherwise, we return the log of the mode of the univariate distribution. Thus, to compute the MAP we simply replace sum nodes with max nodes and return the modes of leaf nodes that have not been instantiated.

Both MAP and probability of evidence are $\mathcal{O}(m)$, where m is the number of edges in the SPN. An SPN generated by `LearnGD` has $n-1$ edges, with n being the number of variables, since such an SPN will always be an SPT. Therefore, inference in an SPT is $\mathcal{O}(n)$.

7. Complexity

From the previous sections, we can finally assess the complexity of our implementation of the `LearnGD` algorithm. We shall rewrite Algorithm 1 in a more detailed form. Algorithm 8 shows a much more in-depth version of the `LearnGD` schema. The idea remains the same: find independencies and similar clusters.

We first check whether the scope $\mathbf{V}$ is composed of one and only one variable. If this is the case, we have reached the recursion base. Let $\{X\} = \mathbf{V}$ be the scope at the base of the recursion. To find the univariate distribution of X from data $\mathbf{D}$, we count the frequencies of each instantiation of X in $\mathbf{D}$. Let us recall our previous notation regarding counting frequencies in data. The function N takes a set of instantiations ν of variables and returns an integer representing the total counts of ν in $\mathbf{D}$. The univariate probability distribution $\mathcal{U}$ derived from each count of X in $\mathbf{D}$ is given by

$$\mathcal{U}[X = x] = \frac{N[X = x]}{N[X = \cdot]}.$$

That is, for each point x in $\mathcal{U}$, its value is the probability of x appearing in $\mathbf{D}$. We apply Laplace smoothing to avoid the zero value. We simply consider each frequency valuation to be $N[X = x] + 1$ and the overall count to be $N[X = \cdot] + |\operatorname{Val}(X)|$, where the function $\operatorname{Val}(X)$ returns the set of all possible valuations of X.

If $\mathbf{V}$ is not a singleton, we find independent partitions with `IndepGraphUF`. If the set of partitions $\mathbf{P}$ has one element, this indicates we have found each variable to be indirectly or directly independent of each other. If we have found independencies, we simply return a new product node whose children are the recursive calls with the different partitions of variables as scope. Otherwise, we proceed to find clusters.

Finding the clusters is done by running `DBSCAN`. If the set of clusters $\mathbf{C}$ is greater than one, we return a new sum node whose children are the recursive calls with $\mathbf{C}_i$ as data and $|\mathbf{C}_i|/|\mathbf{D}|$ as weights. If we have instead found $\mathbf{C}$ to be one big cluster containing all elements, we fully factorize the data. To do this, we create a new product node and for each variable V in $\mathbf{V}$, we create a new univariate probability distribution from V and add it as a child of the product node.

Algorithm 8 `LearnGD`

```
Input  Set $\mathbf{D}$ of instances (data)
Input  Set $\mathbf{V}$ of variables (scope)
Input  Parameters $\mu$ and $\varepsilon$ for the DBSCAN clustering
Output  An SPN representing the probability distribution of $\mathbf{D}$ and $\mathbf{V}$
 1: if $|\mathbf{V}| = 1$ then                                   ▷ Implies $|\mathbf{V}| = 1$
 2:     Let $\lambda$ be a new leaf node
 3:     Let $\{X\} = \mathbf{V}$ be the current scope
 4:     Construct a new distribution $\mathcal{U}$ from each instantiation of $X$ in $\mathbf{D}$
 5:     Assign $\mathcal{U}$ to $\lambda$
 6:     return $\lambda$
 7: end if
 8: ▷ We must now attempt to find independent partitions in $\mathbf{V}$.
 9: Let $\mathbf{P} \leftarrow$ IndepGraphUF$(\mathbf{D}, \mathbf{V})$ the set of independent partitions
10: if $|\mathbf{P}| > 1$ then                  ▷ We have found independent partitions
11:     Let $\pi$ be a new product node
12:     for each $\mathbf{P}_i \in \mathbf{P}$ do
13:         Let $c_i$ be a new child node of $\pi$
14:         $c_i \leftarrow$ LearnGD$(\mathbf{D}, \mathbf{P}_i)$
15:     end for
16:     return $\pi$
17: else
18:     Let $\mathbf{C}gets$ DBSCAN$(\mu, \varepsilon)$ the set of clusters
19:     if $|\mathbf{C}| > 1$ then
20:         Let $\sigma$ be a new sum node
21:         for each $\mathbf{C}_i \in \mathbf{C}$ do
22:             Let $c_i$ be a new child node of $\sigma$
23:             Let $w_i = |\mathbf{C}_i|$ the $i$-th weight of the $i$-th child
24:             $c_i \leftarrow$ LearnGD$(\mathbf{C}_i, \mathbf{V})$
25:             Add weight $w_i$ to edge $e_{\sigma,c_i}$
26:         end for
27:         return $\sigma$
28:     else                    ▷ If data is one big cluster, fully factorize $\mathbf{D}$
29:         Let $\pi$ be a new product node
30:         for each $V_i \in \mathbf{V}$ do
31:             Let $\lambda$ be a new leaf node
32:             Construct a new distribution $\mathcal{U}$ from each instantiation of $V_i$ in $\mathbf{D}$
33:             Assign $\mathcal{U}$ to $\lambda$
34:             Add $\lambda$ as child of $\pi$
35:         end for
36:         return $\pi$
37:     end if
38: end if
```

We already know that the complexity of `IndepGraphUF` is $\mathcal{O}(|\mathbf{V}|^2k^2)$, where $k = \max|\operatorname{Val}(X)|$, and `DBSCAN`'s is $\mathcal{O}(|\mathbf{D}|^2)$. From this, we can easily conclude that

`LearnGD`, in our implementation, has complexity $\mathcal{O}(n^2k^2 + m^2)$, where $n = |\mathbf{V}|$, $m = |\mathbf{D}|$ and $k = \max|\operatorname{Val}(X)|$. Memory usage is $\mathcal{O}(2n + k^2 + m^2)$.

8. Experiments

In this section we show the experiments we have made and the results derived from them. We have evaluated our implementation by applying two classical vision problems: image classification and image completion. We used three datasets for our experiments. The first, named `digits`, is composed of 70 hand drawn PBM images of digits ranging from zero to nine. These images are 20 by 30 pixels wide, with two possible values for each pixel: 0 or 1. If the pixel is set to 0, we color it black. Otherwise, the pixel is white. This dataset was made for hand drawn digit recognition with binary values. The second dataset, `caltech` is based on the Caltech101 object database [FFP14] and was used for image classification. We made slight modifications to the database in order to simplify the tests, as we shall explain later. The third and last database is `olivetti`, a modified Olivetti Faces dataset. This dataset was used for image completion. All dataset images were converted to PGM format and all images in each dataset have same dimensions. We first show and elaborate on the classification results from datasets `digits` and `caltech`. We then talk about the image completion results achieved with dataset `olivetti`. We ran all experiments on a 4-cores 1.8GHz processor with 16GB of RAM.

For all datasets we assumed that each pixel is a variable that takes values according to the gray tone displayed in the image. Each dataset has a different fixed number of maximum values. For instance, dataset `digits` has max value 2, since it is a binary valued (black or white) image, whilst dataset `caltech` has max value 256 (8-bit grayscale). We consider each instance of data $\mathbf{D}$ to be a single image represented by a vector V. Each element v_i in V is the value of a pixel at position $(\lfloor i/w \rfloor, i - \lfloor i/w \rfloor)$, where w is the width of the images.

For classification, we used cross-validation to average the error of each iteration. Let n be the number of images of the dataset $\mathbf{D}$ to be run, and let k be the number of classes in $\mathbf{D}$. We assume that each class has the same number of n/k images. Let $\mathcal{T}$ be the training set, and $\mathcal{S}$ be the test set. We define a real $p \in (0,1)$ that defines the proportion of images sent to each set. That is, the number of images in $\mathcal{T}$ is defined as $\lfloor n/p \rfloor$ and the size of $\mathcal{S}$ as $\lceil n/(1-p) \rceil$. Once we have p set, we randomly select which images go to $\mathcal{T}$ and which to $\mathcal{S}$, and then run the classification job on $\mathcal{T}$ and $\mathcal{S}$. We then repeat this procedure t times and take the average. The training dataset contains $w \times h$ variables representing the pixels, with w and h being the width and height of the image, and one more variable to represent the class label (e.g. the digit it represents).

We used nine different p values ranging from $p = 0.1$ to $p = 0.9$ with 0.1 step increments. For each of these values, we fed the SPN the train set and ran classification on the test set. Each test set instance contains $w \times h$ variables representing the pixels of the test image. We then maximize the probability of finding such pixel instantiations. That is, we wish to find

$$\arg\max_x \Pr(X = x|\mathbf{E}) = \arg\max_x \frac{\Pr(X = x, \mathbf{E})}{\Pr(\mathbf{E})}$$

where $\mathbf{E}$ are the values of the pixels what we have observed from our test image. If the argument x that maximizes such probability matches the real class label, then we have classified correctly. For each of these p values, we iterated 10 times with different random training and test sets such that they obey p. This avoids skewed results due to chance.

The `digits` dataset is comprised of 700 hand drawn images of digits ranging from zero to nine. Each class is a digit and each class has 70 instances of 20×30 images. Images have a binary value for each pixel, zero is set as black and one as white.

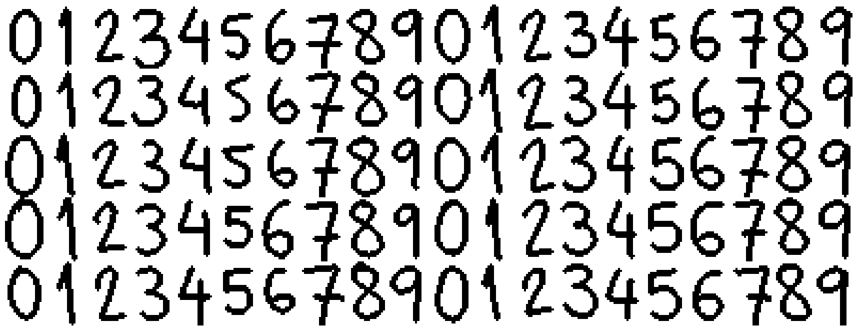


FIGURE 2. A sample of the `digits` dataset.

Figure 3 shows the percentages of correct classifications on the `digits` dataset. Averaging all the right classifications in all p values yields a 98.249% hit, with 30919 images being correctly classified out of 31470. We also measured the time it took for each run of 10 iterations to complete. The results are shown in Figure 4. An interesting point of inflection occurs when $p \geq 0.5$. We conjecture that this occurs because learning the structure with this dataset partition is faster than running each $\Pr(X = x|\mathbf{E})$ and finding the maximum. When $p > 0.5$, the train set size becomes greater than the test set size. When the dataset is sufficiently simple (as is the case of `digits`, where $\text{Val}(X) = 2$ for X a pixel variable) and sufficiently big, learning time remains fairly short whilst computing each probability of x given the test image pixels takes even less time at each increment of p. As we shall see in the other datasets, this phenomena is highly dependent on data.

`LearnGD` has memory usage of $\mathcal{O}(2n+k^2+m^2)$, whilst computing the probability of evidence uses no significant memory (we store the logs to avoid recomputing, at most) which means the expected behavior of the maximum memory used in each run should always increase. This is shown in Figure 5, where we can see that a simple dataset of 700 20×30 images can take up to ≈ 160 MB.

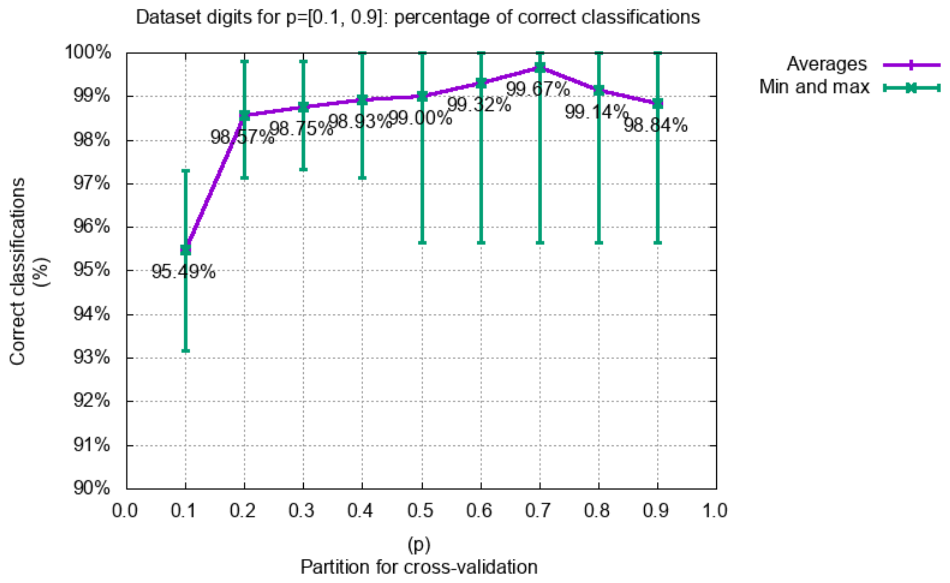


FIGURE 3. This graph shows the percentages of correct hits when classifying the `digits` dataset. Each classification run is composed of 10 randomly selected training and testing sets iterations. On each iteration, we tried to guess the class label of each test image based only on the pixel values. The blue line in Figure 3 shows the average percentage of correct labeling for each p. The errorbars on each p value show the minimum and maximum percentages of each run.

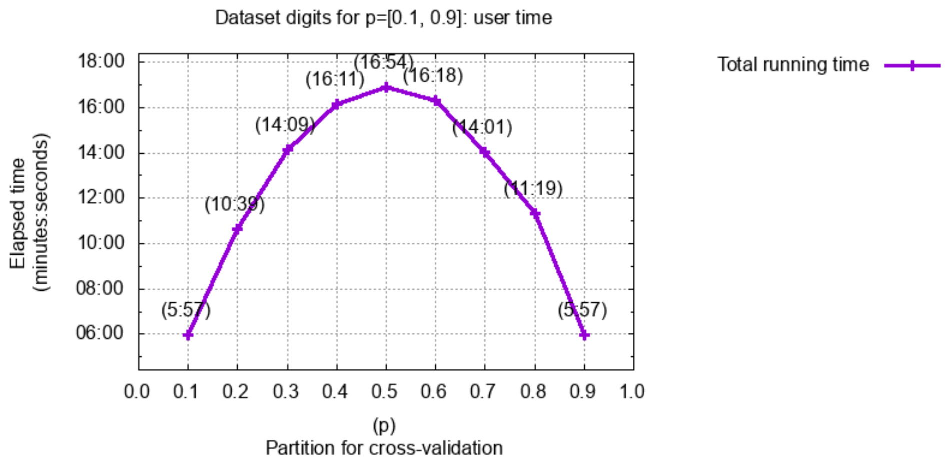


FIGURE 4. This graph shows the elapsed time of each run in minutes and seconds. The point of inflection occurs when the training set is greater than the test set $p > 0.5$.

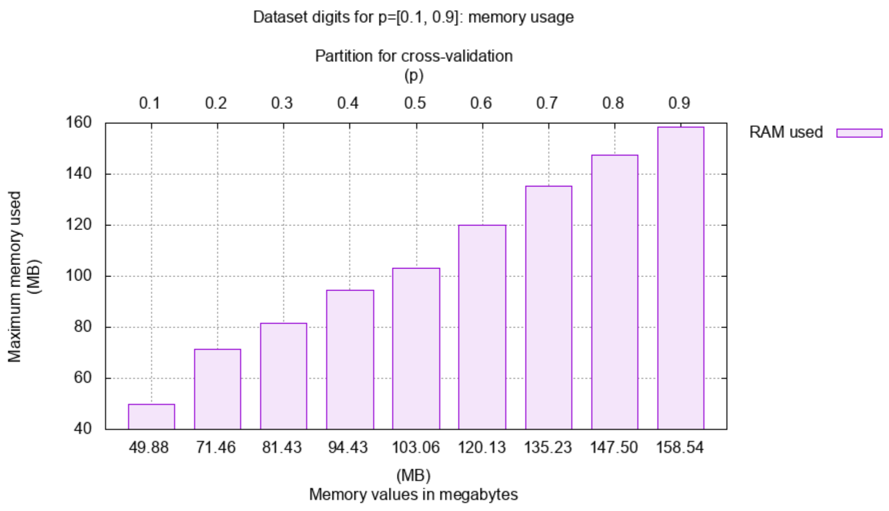


FIGURE 5. This histogram shows the maximum memory used on the `digits` dataset at each run. It gives us an approximate upper bound on how much memory we use in each iteration of a certain value of p. The more p grows, the higher the amount of memory used (as expected).

The `caltech` dataset is a modified Caltech-101 [FFP14] image database. We attemped to recreate the classification experiment done on [PD11] with three Caltech-101 categories: faces, motorbikes and cars. Due to memory constraints, we selected only 100 images per category. These images were then normalized to 150×65 pixels images and converted to grayscale.

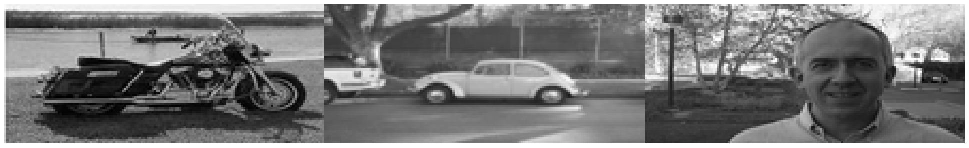


FIGURE 6. A sample of the `caltech` dataset.

Our results are much worse than those achieved in [PD11]. This may be due to our memory constraints, as we used fewer images (300 in total), whereas the original dataset has over 2700 images in total. Another important observation when comparing the two algorithms is the structure learned. In [PD11], the parameters and not the structure is learned, however the structure is made specifically for vision problems, in contrast with `LearnGD`'s general nature. Furthermore, the parameters to clustering (in this case DBSCAN) can yield very different results depending on their values. We fixed the μ and ε parameters for DBSCAN on all datasets. Better results may be achieved by altering these parameters in accordance to the dataset's characteristics.

The overall percentage of correct classifications was 78.078%, lasting at most over 8 hours and 30 minutes to complete a full run of 10 iterations. The maximum amount of memory used in a run was 12.63GB.

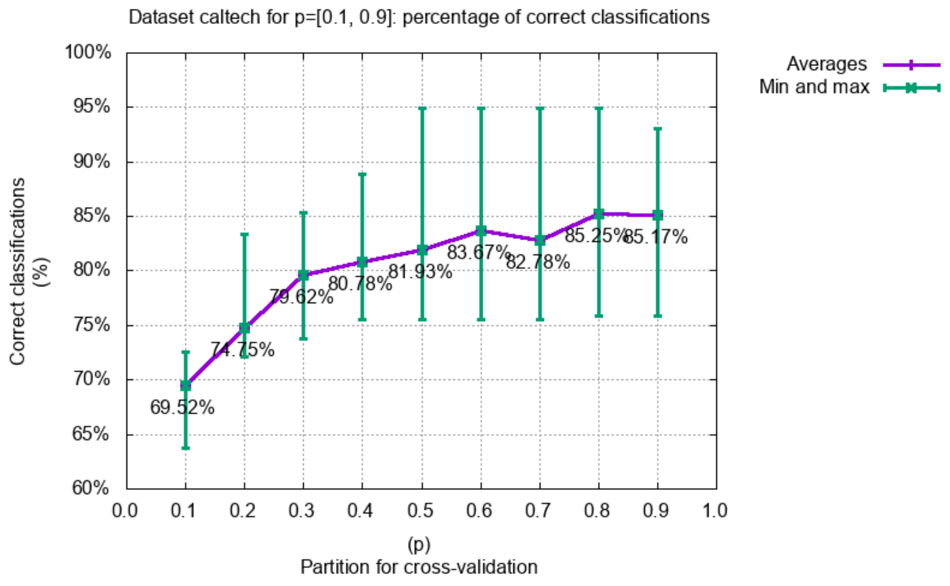


Figure 7. The percentages of correct classifications on each run of 10 iterations for different varying p values. Values varied from 63% to 95%. The average hits are shown by the blue line, with green errorbars showing minimum and maximum values at each run.

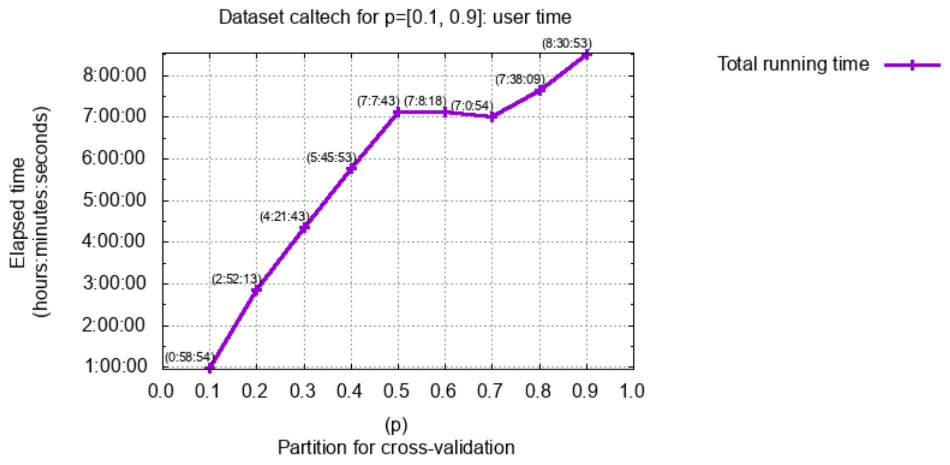


Figure 8. The maximum elapsed time at each run. Values ranged from about 59 minutes up to 8 hours and 31 minutes. Considering each run had 10 iterations, each iteration at $p = 0.1$ lasted close to 6 minutes each, whilst each iteration with $p = 0.9$ lasted about 51 minutes.

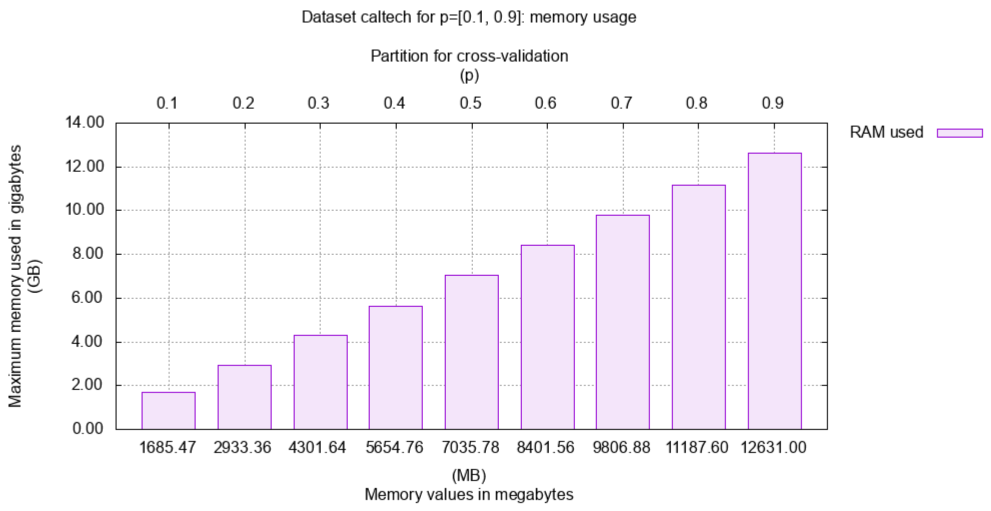


FIGURE 9. The maximum amount of memory used on an iteration at a certain p run. Memory usage took about 12GB at most. Since we were running all experiments on a 16GB machine, we had to cap our memory usage to 16GB. This meant shrinking and capping the total number of images used on our classifications to 300 images.

FIGURE 10. A sample of the original Olivetti dataset (left) with 256 possible pixel values.

The Olivetti Faces dataset is composed of 400 faces of 40 different people, with 10 images for each person, with these ten images portraying different expressions and head positionings. We converted the original dataset into individual 46×56 PGM images with pixel values ranging from $[0, 255]$. We used the entirety of the dataset for the experiments. However, once we converted the original image into a 256-PGM file, the distribution of grayscale values proved to be less than uniform as Figure 12 shows. The low count of values on the extremes of the histogram causes incorrect variable independence results. We then normalized all images to pixel values ranging $[0, 7]$, which caused the counts to become less extreme as Figure 13 shows.

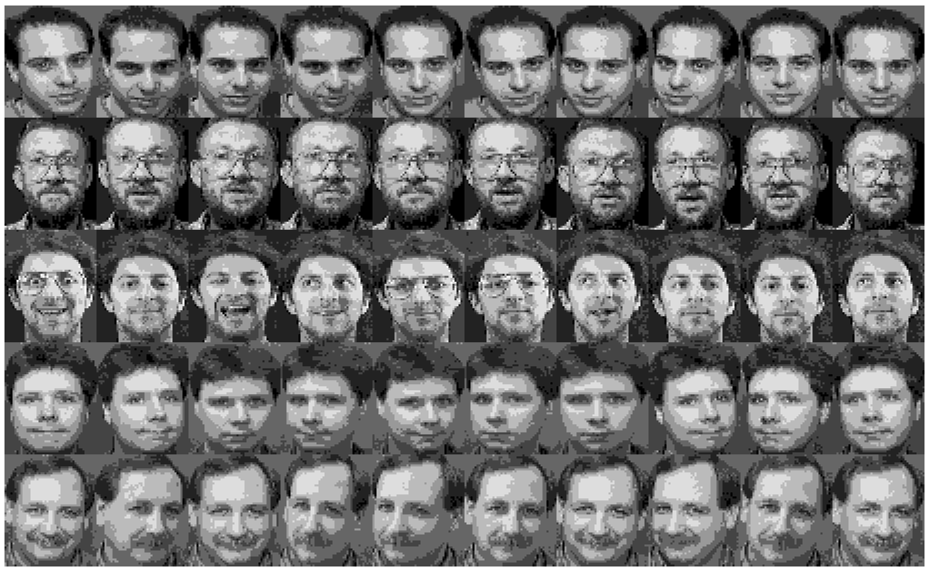


FIGURE 11. A sample of the `olivetti` dataset, where the images in Figure 10 were normalized to $[0, 7]$ possible values. There isn't much loss of facial features on our modified dataset, although it is clear that certain details were indeed lost.

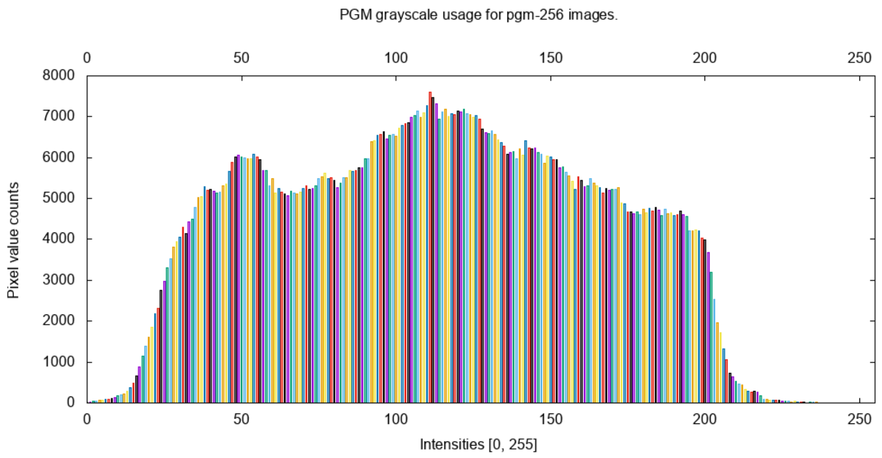


FIGURE 12. The histogram showing the frequencies of each pixel value in all images in the original Olivetti faces dataset. We can clearly see that both extremes have a very low frequency. This is problematic to our variable independence algorithm.

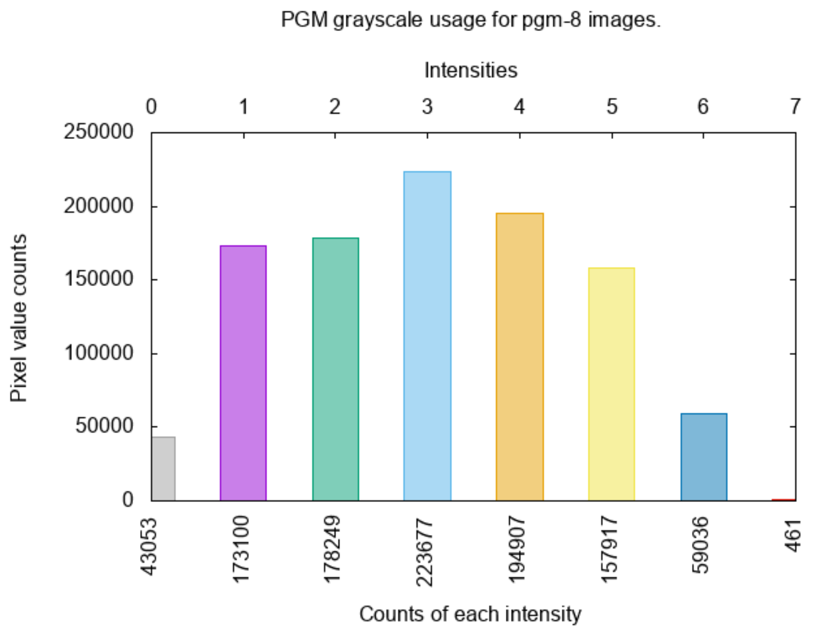


FIGURE 13. The histogram for the normalized Olivetti dataset. This is the histogram for our modified `olivetti` dataset, where pixel values only range from 0 to 7. Comparing to Figure 12 we see that the count of both 0 and 7 values are much higher relative to the rest.

We ran image completion on the `olivetti` dataset. For this task, we had two different sets of test settings. In the first we chose an image of `olivetti` and learned all the rest of the dataset. This meant we had prior knowledge on the facial structure of the person whose image we chose. This obviously resulted in better completion for the image in question. We shall call this completion a C1 completion. In the second C2 setting, we chose a person and trained the SPN only with the images of other people. We then performed image completion on an image of the person whose images were not in the learning set. This means we had no prior knowledge of that face, and based all completions on other people's faces. This is closer to a real-world application, as we do not have a database on every face in the world. Results varied according to each face. As we can see from the resulting images, the SPN had trouble dealing with glasses, beards and wrinkles. It also showed that it can succesfully recognize, in most cases, whether the face is slightly turned to the left or to the right, as well as identify smiles and open mouthes.

For these image completions, we set $\mu = 4.0$, $\varepsilon = 4.0$ for DBSCAN and set a significance value of 0.001 for the variable independence test. These same parameters were used for image classification in the other datasets. We did not test whether other parameter settings would yield better results. One can test these different parameter settings by using our implementation source code [Geh16]. DBSCAN is highly dependent on the instances it is run on. This is because of the nature of each image. For instance, for `caltech` we have clear features that are common for each image (e.g. a motorbike on the foreground), whilst for `olivetti` these features are less obvious (e.g. the outline of glasses or the darker tones of a moustache). For these reasons, we can never have a global all-fitting parameter setting.

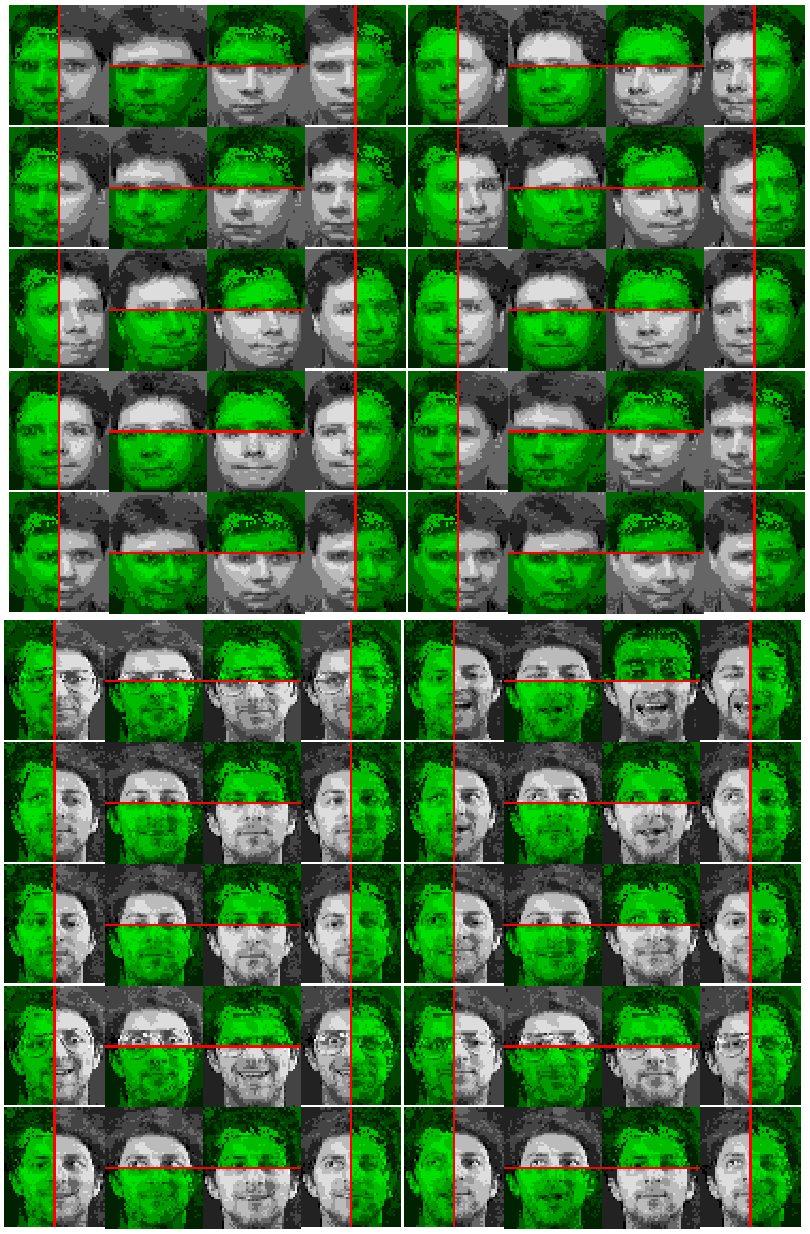


Figure 14. Two C1 image completions on the `olivetti` dataset. The green portion of each image is the completion done by the SPN. The other gray portion is the original image. Each collection of four images are the completions done for the left, bottom, top and right halves of the same image.

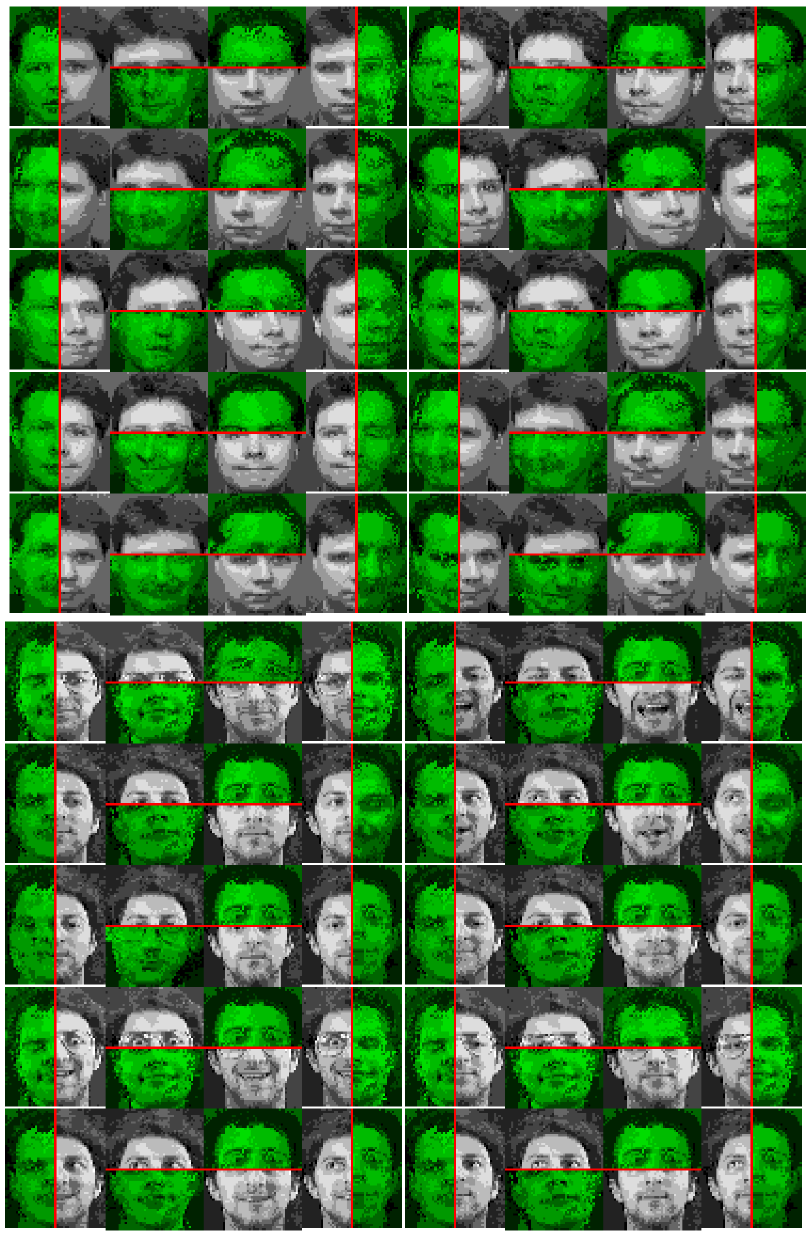


Figure 15. Two C2 image completions on the `olivetti` dataset. When compared to Figure 14, the above results are much less accurate. It is evident the presence of features that appear on other faces, such as a different shaped nose, the presence of a moustache or glasses.

9. Conclusions

References

[DB11] Olivier Delalleau and Yoshua Bengio. "Shallow vs. Deep Sum-Product Networks". In: *Advances in Neural Information Processing Systems 24 (NIPS 1011)* (2011).

[DV12] Aaron Dennis and Dan Ventura. "Learning the Architecture of Sum-Product Networks Using Clustering on Variables". In: *Advances in Neural Information Processing Systems* 25 (2012).

[Est+96] Martin Ester et al. "A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD-96 Proceedings* (1996).

[FFP14] L. Fei-Fei, R. Fergus, and P. Perona. "Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories". In: *IEEE CVPR-2014 Workshop on Generative-Model Based Vision* (2014).

[For65] E. W. Forgy. "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". In: *Biometrics* 21.3 (1965), pp. 761–777. ISSN: 0006341X, 15410420. URL: `http://www.jstor.org/stable/2528559`.

[GD13] Robert Gens and Pedro Domingos. "Learning the Structure of Sum-Product Networks". In: *International Conference on Machine Learning* 30 (2013).

[Geh16] Renato Lui Geh. *GoSPN: an implementation of sum-product networks in Go*. 2016. URL: `https://github.com/RenatoGeh/gospn`.

[PD11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Uncertainty in Artificial Intelligence* 27 (2011).

[Peh+15] Robert Peharz et al. "On Theoretical Properties of Sum-Product Networks". In: *International Conference on Artificial Intelligence and Statistics 18 (AISTATS 2015)* (2015).

[RL14] Amirmohammad Rooshenas and Daniel Lowd. "Learning Sum-Product Networks with Direct and Indirect Variable Interactions". In: *International Conference on Machine Learning 31 (ICML 2014)* (2014).